COMP26120: Algorithms and Imperative Programming

Introducing Complexity

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2017–18
You need this book:

- Make sure you use the up-to-date edition. It is available on the course materials page:
  
  http://studentnet.cs.manchester.ac.uk/ugt/2016/COMP26120/syllabus/

- Read Ch. 1 (pp. 1–50).
- Pay particular attention to:
  - Pseudocode
  - Big-O notation and its relatives
  - The mathematical basics.
- Also read pp. pp. 689–690 and 695–696.
Outline

Getting started: two ways of computing variance

Big-O notation

Some details: What is an operation, and how big is a number?

Example: Euclid’s algorithm for finding highest common factors

Example: powers in modular arithmetic
• Let us begin with a simple example.

• Suppose we have a collection of numbers \(x_1, \ldots, x_n\), and want to compute the variance, defined by the formula:

\[
\sigma^2 = \frac{1}{n^2} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(x_i - x_j\right)^2 = \frac{1}{2n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \left(x_i - x_j\right)^2.
\]

• We could just do it:

\[
\text{var1}(x_1, \ldots, x_n) \\
\quad s := 0 \\
\quad \text{for } i \text{ from } 1 \text{ to } n - 1 \\
\quad \quad \text{for } j \text{ from } i + 1 \text{ to } n \\
\quad \quad \quad s := s + (x_i - x_j)^2 \\
\quad \text{return } s/n^2 \\
\text{end}
\]
• To see why this wouldn't be a good idea, let's count how much work is done.

\[
\text{var1}(x_1, \ldots, x_n)
\]
\[
s := 0
\]
\[
\text{for } i \text{ from 1 to } n - 1
\]
\[
\quad \text{for } j \text{ from } i + 1 \text{ to } n
\]
\[
\quad s := s + (x_i - x_j)^2
\]
\[
\text{return } s/n^2
\]
\[
\text{end}
\]

• We do \( \sum_{i=1}^{n-1} (n - i) = \sum_{i=1}^{n-1} i = \frac{1}{2} (n - 1)n \) executions of the line \( s := s + (x_i - x_j)^2 \) plus one final squaring and division—about \( \frac{3}{2} (n - 1)n + 2 \) operations.
But suppose you notice that the variance of $x_1, \ldots, x_n$ is actually the mean squared distance from the mean, $\mu$. Noting that $\mu = \sum_{i=1}^{n} x_i / n$:

$$
\sigma^2 = \sum_{i=1}^{n-1} \sum_{i=i+1}^{n} (x_i - x_j)^2 / n^2 = \sum_{i=1}^{n} (x_i - \mu)^2 / n.
$$

Then the following algorithm will then work:

```plaintext
var2(x_1, \ldots, x_n)
  m := 0
  for i from 1 to n
    m := m + x_i
    m := m/n  \% m now holds the mean
  s := 0
  for i from 1 to n
    s := s + (x_i - m)^2
  return s/n
end
```
• Now let’s see how much work was done again:
  \[
  \text{var2}(x_1, \ldots, x_n)
  \]
  \[
  m := 0 \\
  \text{for } i \text{ from } 1 \text{ to } n \\
  \quad m := m + x_i \\
  m := m/n \quad \% \ m \text{ now holds the mean} \\
  s := 0 \\
  \text{for } i \text{ from } 1 \text{ to } n \\
  \quad s := s + (x_i - m)^2 \\
  \text{return } s/n
  \]
  
  • Here we do \( n \) additions in the first loop, and \( n \) subtractions, squarings and additions in the second loop, plus one division after each loop, making \( 4n + 2 \) operations, much less (for large \( n \)) than \( \frac{3}{2}(n - 1)n + 2 \).
• Observe
  • Algorithms are given in pseudocode.
  • The correctness of the algorithm \( \text{var2} \) needs to be established. Specifically, we have to prove that

\[
\frac{1}{n^2} \sum_{i=1}^{n-1} \sum_{i=i+1}^{n} (x_i - x_j)^2 = \frac{1}{n} \sum_{i=1}^{n} \left( x_i - \frac{1}{n} \sum_{i=1}^{n} x_i \right)^2.
\]

• We could quibble endlessly about exactly how many operations are involved in these algorithms, but we’d rather not . . .
• Such quibbles are irrelevant, because \( \text{var2} \) is clearly superior to \( \text{var1} \).
• This lecture is about how to articulate these ideas.
Outline

Getting started: two ways of computing variance

Big-O notation

Some details: What is an operation, and how big is a number?

Example: Euclid’s algorithm for finding highest common factors

Example: powers in modular arithmetic
• When comparing growth-rates of functions, it is often useful to ignore
  • small values
  • linear factors
• That is, we are interested in how functions behave in the long run, and up to a linear factor.
• This is essentially to enable us to abstract away from relatively trivial implementation details.
• The main device used for this is big-O notation. If \( f : \mathbb{N} \rightarrow \mathbb{N} \) is a function, then \( O(f) \) denotes the set of functions:

\[
\{ g : \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 \in \mathbb{N} \text{ and } c \in \mathbb{R}^+ \text{ s.t. } \forall n > n_0, g(n) \leq c \cdot f(n) \}.
\]

• Thus, \( O(f) \) denotes a set of functions.
• To see why this is useful, consider the sets of functions

\[ O(n) = \{ g : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N}, \ c \in \mathbb{R}^+ \text{ s.t. } \forall n > n_0, \ g(n) \leq cn \} \]

\[ O(n^2) = \{ g : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N}, \ c \in \mathbb{R}^+ \text{ s.t. } \forall n > n_0, \ g(n) \leq cn^2 \}. \]

• The following should now be obvious:
  • The function \( g_2(n) = 4n + 2 \) is in \( O(n) \).
  • The function \( g_1(n) = \frac{3}{2}(n - 1)n + 2 \) is in \( O(n^2) \).
  • The function \( g_1(n) \) is not in \( O(n) \).

• Notice, of course, that \( O(n) \subsetneq O(n^2) \).
• So now we can express succinctly the difference between running times of our algorithms var1 and var2:
  • The running time of var1 is in $O(n^2)$ (but not in $O(n)$);
  • The running time of var2 is in $O(n)$.
• Often, we forget that $O(f)$ is technically a set of functions, and say:
  • The running time of var1 is $O(n^2)$ (or: is order $n^2$);
  • The running time of var2 is $O(n)$ (or: is order $n$).
But this is really just a manner of speaking.
• Of course, you can have $O(f)$ for any $f : \mathbb{N} \to \mathbb{N}$:
  - $O(\log n)$
  - $O(\log^2 n)$
  - $O(\sqrt{n})$
  - $O(n), O(n^2), O(n^3), \ldots$
  - $O(2^n), O(2^{n^2}), \ldots$
  - $O(2^{2^n}), O(2^{2^{2n}}), \ldots$
• Sometimes you will see other asymptotic measures:

• If $f : \mathbb{N} \rightarrow \mathbb{N}$ is a function, then $\Omega(f)$ denotes the set of functions:

\[
\{ g : \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 \in \mathbb{N} \text{ and } c \in \mathbb{R}^+ \text{ s.t. } \forall n > n_0, g(n) \geq c \cdot f(n) \}.
\]

• Thus, $g \in \Omega(f)$ states that, asymptotically, $g$ grows as fast as $f$.

• $f \in \Omega(g)$ if and only if $g \in O(f)$.

• People also sometimes write

\[
\Theta(f) = O(f) \cap \Omega(f).
\]

• To say that $f \in \Theta(g)$ is to say that asymptotically, $f$ and $g$ grow as fast as each other.
• **To think about:**
  
  • Make sure you understand why \( f(n) \leq g(n) \) for all \( n \) implies \( O(f) \subseteq O(g) \).
  
  • Why do you not hear people talking about \( O(6n + 7) \)?
  
  • Show that \( f \in \Omega(g) \) just in case \( g \in O(f) \).
  
  • Give a succinct but accurate characterization of \( O(1) \) in plain English.
Outline

Getting started: two ways of computing variance

Big-O notation

Some details: What is an operation, and how big is a number?

Example: Euclid’s algorithm for finding highest common factors

Example: powers in modular arithmetic
We said that the time-complexity of var2 is in $O(n)$, but what, exactly, does this mean?

Answer: to say that an algorithm $A$ runs in time $g$ means the following.

*Given an input of size $n$, the number of operations executed by $A$ is bounded above by $g(n)$.*

This raises two important issues:

- What is an operation?
- How do we measure the size of the input?
• Deciding what to count as an operation is a bit of a black art. It depends on what you want your analysis for.
• For most practical applications, it is okay to take the following as operations:
  • arithmetic operations (e.g. +, *, /, %) on all the basic number types)
  • assignments (e.g. a := b, a[i] = t, t = a[i])
  • basic tests (e.g. a = b, a ≥ b)
  • Boolean operations (e.g. &, !, ||).
• Things like allocating memory, managing loops are often ignored—again, this may depend on the application.
• Note that, for some applications, this accounting régime might be misleading.
• Imagine, for example, an cryptographic algorithm requiring to perform arithmetic on numbers hundreds of digits long.
• In this case, we would probably want to count the number of logical operations involved.
• For example, to multiply numbers with $p$ bits and $q$ bits, we require in general about $pq$ logical operations.
• There is a formal model of computation, the Turing Machine, which specifies precisely what counts as a basic operation.
• But in this course, we shall not use the Turing machine model.
• The question of how to measure the size of the input is rather trickier.

• Officially, the input to an algorithm is a string.

• Often, that string represents a number, or a sequence of numbers, but it is still a string.

• What is the size of the following inputs?
  • The cat sat on the mat
  • 1
  • 13
  • 445
  • 65535

• The size of a positive integer \( n \) (in canonical decimal representation) is \( \lceil \log_{10} n \rceil + 1 \), not: \( n \).
Outline

Getting started: two ways of computing variance

Big-O notation

Some details: What is an operation, and how big is a number?

Example: Euclid’s algorithm for finding highest common factors

Example: powers in modular arithmetic
- Suppose you want to compute the highest common factor (hcf) of two non-negative integers $a$ and $b$.
- Note that the hcf is sometimes called the greatest common divisor (gcd).
- Assume $a \geq b$. A little thought shows that, letting $r = a \mod b$, we have, for some $q$

\[
a = qb + r
\]

\[
r = qb - a
\]

so that the common factors of $a$ and $b$ are the same as the common factors of $b$ and $r$. Hence:

\[\text{hcf}(a, b) = \text{hcf}(b, r).\]
This gives us the following very elegant algorithm for computing highest common factors.

\[
\text{hcf}(a, b) \quad \text{(Assume } 0 \neq a \geq b) \\
\text{if } b = 0 \\
\quad \text{return } a \\
\text{else} \\
\quad r := a \mod b \\
\quad \text{return } \text{hcf}(b, r) \\
\text{end}
\]

This is so simple, it hurts.
• How long does \( \text{hcf}(a, b) \) take to run?

Well, let \( a_1, a_2, \ldots, a_\ell \) be the first arguments of successive calls to \( \text{hcf} \) in the computation of \( \text{hcf}(a, b) \). (Thus, \( a = a_1 \).)

Certainly \( a_1 > a_2 > \cdots > a_\ell \), so the algorithm definitely terminates.

Assuming \( \ell > 2 \), consider \( h \) in the range \( 1 \leq h \leq \ell - 2 \). If \( a_{h+1} \leq a_h/2 \), then \( a_{h+2} < a_h/2 \). On the other hand, if \( a_{h+1} > a_h/2 \), then \( a_{h+2} = a_h \mod a_{h+1} < a_h/2 \).

Either way, \( a_{h+2} < a_h/2 \). So the number of iterations is at most \( \max(2, 2\lceil \log_2 a \rceil) \). That is, the algorithm is linear in the size of the input. (Actually, the algorithm performs slightly better than this.)
Electronic Digital Computers

A small electronic digital computing machine has been operating successfully for some weeks in the Royal Electronic Computing Machine Laboratory, which is at present housed in the Electrical Engineering Department of the University of Manchester. The machine is purely experimental, and is on too small a scale to be of commercial value. It was built primarily to test the soundness of the storage principle employed and to permit experience to be gained with this type of machine before embarking on the design of a full-scale machine. However, apart from its small size, the machine is, in principle, "universal" in the sense that it can be used to solve any problem (subject to the limitations of its programming in elementary instructions); the programme can be changed without any mechanism or electronic circuit changes.

The essential parts of such a machine are: (1) a store for information and orders; (2) various arithmetical organs (for example, adder, multiplier); (3) a control unit.

The present machine contains the minimum set of facilities for a universal machine, namely: (a) if A is any member in the store, it can be written into a central "accumulator" A; (b) A can be subtracted from what is in A; (c) The number A can be written in an assigned address in the store. (6) The answer (in A) or a direct writing of A can be used for the following operations: (a) addition (in A); (b) subtraction (by use of the store); (c) comparison of the numbers in the store; (d) conditional operations according to the store contents; (e) the control of the machine; (f) the control of the machine.

Excitation Probability Functions of Atomic and Molecular Energy Levels

In the correlative of the electronic and spectroscopic properties of a given discharge in a molecular or atomic atmosphere, the knowledge of the intensity of the excitation of the gas atoms becomes important. In all but the simplest cases, no more than a rough analysis can be attempted with the present scanty experimental and theoretical knowledge of the interaction of slow electrons with atoms and molecules. The purpose of this communication is to present an empiric formula for the excitation probability functions of the atomic and molecular levels which seem to fit the few experimental curves fairly closely, and at least provides a basis for calculation where previously only generalised discussion was possible.

In principle, the excitation probability functions of atoms or molecules may be calculated from quantum mechanics. However, in actual fact, very few atoms or molecules have been considered. It is found that, for the problem of obtaining the probability of exciting the gas atoms, the interaction of the gas atoms with slow electrons-the interesting case from the point of view of glow discharges, when the approximation becomes invalid. Apart from these difficulties, it is necessary that the wave functions of the ground and excited states should be known. The wave functions of very few of the energy states of atoms and molecules have, in fact, been treated. The quantum-mechanical method does not express the excitation probability function in a form which is readily applicable to discharge problems. Blackett's work is an interesting semi-chemical approach to the subject using the principles of conservation of energy and momentum. This, however, was limited and approximate.

The shapes of excitation functions observed experimentally for single and multiple excited states from a single ground state are different. The singlet excitation function starts from zero as the energy of the level concerned is a broad maximum somewhere above it, while the triplet excitation function rises from zero at the energy of the level concerned to a
Outline

Getting started: two ways of computing variance

Big-O notation

Some details: What is an operation, and how big is a number?

Example: Euclid’s algorithm for finding highest common factors

Example: powers in modular arithmetic
• Recall the definition of \( m \mod k \), for \( k \) an integer greater than 1:

\[
17 \mod 6 = 5 \\
14 \mod 2 = 0 \\
117 \mod 10 = 7
\]

• When performing arithmetic mod \( k \), we can stay within the numbers 0, \ldots, \( k - 1 \):

\[
17 + 5564 \mod 10 = 7 + 4 \mod 10 = 11 \mod 10 = 1 \\
17 \cdot 5564 \mod 10 = 7 \cdot 4 \mod 10 = 28 \mod 10 = 8 \\
5564^{17} \mod 10 = 4^{17} \mod 1 = 17179869184 \mod 10 = 4
\]

• Modular arithmetic is important in cryptography.
• Recall the definition of $m \mod k$, for $k$ an integer greater than 1:

$$17 \mod 6 = 5$$
$$14 \mod 2 = 0$$
$$117 \mod 10 = 7$$

• When performing arithmetic mod $k$, we can stay within the numbers $0, \ldots, k - 1$:

$$17 + 5564 \mod 10 = 7 + 4 \mod 10 = 11 \mod 10 = 1$$
$$17 \cdot 5564 \mod 10 = 7 \cdot 4 \mod 10 = 28 \mod 10 = 8$$
$$5564^{17} \mod 10 = 4^{17} \mod 1 = 17179869184 \mod 10 = 4$$

• Modular arithmetic is important in cryptography.
• Modular arithmetic is particularly nice when the modulus is a prime number, $p$.

• If $1 \leq a < p$, then there is a unique number $b$ such that

$$a \cdot b = b \cdot a = 1 \mod p.$$ 

In that case we call $b$ the inverse of $a$ (modulo $p$) and write $b = a^{-1}$.

• For example,

$$3 \cdot 5 = 5 \cdot 3 = 1 \mod 7,$$

so 3 and 5 are inverses modulo 7.
• Here is an algorithm to compute \( a^b \mod k \).
  (Note that we may as well assume that \( a < k \).)

\[
pow1(a, b, k) \\
  s := 1 \\
  \text{for } i \text{ from 1 to } b \\
  s := s \cdot a \mod k \\
  \text{return } s \\
\]

• The number of operations performed here is clearly \( O(b) \).
• Therefore the time complexity is . . . .
• Here is an algorithm to compute $a^b \mod k$. (Note that we may as well assume that $a < k$.)

$$\text{pow1}(a,b,k)$$

$$s := 1$$

for $i$ from 1 to $b$

$$s := s \cdot a \mod k$$

return $s$

end

• The number of operations performed here is clearly $O(b)$.

• Therefore the time complexity is $O(2^n)$—i.e. exponential.
Here is an algorithm to compute \( a^b \mod k \).
(Note that we may as well assume that \( a < k \).)

\[
\text{pow1}(a, b, k) \\
\quad s := 1 \\
\quad \text{for } i \text{ from } 1 \text{ to } b \\
\quad \quad s := s \cdot a \mod k \\
\quad \text{return } s 
\]

- The number of operations performed here is clearly \( O(b) \).
- Therefore the time complexity is \( O(2^n) \)—i.e. exponential.
- That’s right, exponential, not linear: the size of the input \( b \) is \( \log b \). (Note that \( a \) and \( k \) don’t really matter here.)
- Reminder: \( 2^{\log_2 n} = n \).
• Here is a better algorithm to compute \( a^b \mod k \).

\[
\text{pow2}(a, b, k) \\
\begin{align*}
    &d := a, \ e := b, \ s := 1 \\
    \text{until} \ &e = 0 \\
    &\quad \text{if} \ e \ \text{is odd} \\
    &\quad \quad s := s \cdot d \ \text{mod} \ k \\
    &\quad d := d^2 \ \text{mod} \ k \\
    &\quad e := \lfloor e/2 \rfloor \\
    \text{return} \ &s \\
\end{align*}
\]

• The number of operations performed here is proportional to the number of times \( d = b \) can be halved before reaching 0, i.e. at most \( \lceil \log_2 b \rceil \). Thus, this algorithm has running time in \( O(n) \), i.e. \text{linear} in the size \( n \) of the input \( b \). (Again, \( a \) and \( k \) don’t really matter here.)
• To see how this number works, think of $b$ in terms of its binary representation $b = b_{n-1}, \ldots, b_0$, i.e.

$$b = \sum_{h=0}^{n-1} b_h \cdot 2^h,$$

so that

$$a^b = \prod_{h=0}^{n-1} a^{(b_h \cdot 2^h)}$$

And of course

$$a^{(b_h \cdot 2^h)} = \begin{cases} 1 & \text{if } b_h = 0 \\ a^{(2^h)} & \text{if } b_h = 1. \end{cases}$$

But the variable $e$ holds $a^{2^h}$ on entry to the $h$th iteration on the loop (counting from $h = 0$ to $h = n - 1$).
• Compute $7^{11} \mod 10$. (N.B. $7^{11}$ is actually 1977326743.)

Before loop:

$s \leftarrow 1$
$d \leftarrow 7$
$e \leftarrow 11 (= \text{Binary } 1011)$

Round 1:

$s \leftarrow 7$ ($e$ is odd and $1 \cdot 7 = 7 \mod 10$)
$d \leftarrow 9$ ($7^2 = 9 \mod 10$)
$e \leftarrow 5$

Round 2:

$s \leftarrow 3$ ($e$ is odd and $7 \cdot 9 = 3 \mod 10$)
$d \leftarrow 1$ ($7^4 = 9^2 = 1 \mod 10$)
$e \leftarrow 2$

Round 3:

$s \leftarrow 3$ ($e$ is even)
$d \leftarrow 1$ ($7^8 = 1^2 = 1 \mod 10$)
$e \leftarrow 1$

Round 4:

$s \leftarrow 3$ ($e$ is odd and $3 \cdot 1 = 1 \mod 10$)
$d \leftarrow 1$ ($7^{16} = 1^2 = 1 \mod 10$)
$e \leftarrow 0$

• At some point $d$ became 1. Do you see an optimization?
• Raising positive numbers to various powers modulo $k$ produces 1 more often than you think.
• This is of special interest when $k$ is some prime number, $p$.
• For example, set $k = p = 7$. In the following, all calculations are performed modulo 7.

$$
\begin{align*}
1^1 &= 1 \\
2^1 &= 2 & 2^2 &= 4 & 2^3 &= 1 \\
3^1 &= 3 & 3^2 &= 2 & 3^3 &= 6 & 3^4 &= 4 & 3^5 &= 5 & 3^6 &= 1 \\
4^1 &= 4 & 4^2 &= 2 & 4^3 &= 1 \\
5^1 &= 5 & 5^2 &= 4 & 5^3 &= 6 & 5^4 &= 2 & 5^5 &= 3 & 5^6 &= 1
\end{align*}
$$

• Notice that, for $1 \leq a < p$, the smallest $k$ such that $a^k \equiv 1 \pmod{p}$ divides $p - 1$ (this is always true for $p$ prime).
• Hence $a^{p-1} \pmod{p}$ is always 1 (this is Fermat’s little theorem).
• However, there is always some number $a$ such that the various powers $g^i$ cover the whole of $\{1, 2, \ldots, p - 1\}$ ($g$ is a primitive root modulo $p$).
• Let $p$ be a prime, and consider the equation

$$a^x = y \mod p.$$ 

• If $a$ is a primitive root modulo $p$, then, for every $y$ $(1 \leq y < p)$, such an $x$ $(1 \leq y < p)$ exists.

• In that case, the number $x$ is called the discrete logarithm of $y$ with base $a$, modulo $p$.

• Thus, the discrete logarithm is an inverse of exponentiation.

• We have seen that, for fixed $a$ and $p$, computing

$$y = a^x \mod p$$

for a given $x$ is very fast. However, no such fast algorithm is known for recovering $x$ from $y$.

• That is: modular exponentiation may be an example of a one-way function—easy to compute, hard to invert.
• Such one-way functions can be used for cryptography.
• Fix a prime, $p$ a primitive root $g$ modulo $p$.
• Choose a **private key**: $x$ ($1 \leq x < p - 1$).
• Broadcast the **public key**: $(p, g, y)$, where $y = g^x$.
• Suppose someone wants to send you a message $M$ (assume $M$ is an integer $1 \leq M < p$).
• He picks $k$ relatively prime to $p - 1$, sets

$$a \leftarrow g^k \mod p$$

$$b \rightarrow M y^k \mod p$$

and sends the ciphertext $C = (a, b)$.
• To decode $C = (a, b)$, you set

$$M' \leftarrow b/(a^x) \mod p.$$
• To see that you get the proper message:

\[ M' = \frac{b}{a^x} \mod p = M y^k (a^x)^{-1} \mod p \]
\[ = M (g^{xk})(g^{xk})^{-1} \mod p \]
\[ = M \]

• To see that this is secure, notice that to encode, one needs the public key \( y = g^x \), but to decode, one needs the private key \( x \) (which only you have).

• In other words, to break the code, an enemy needs to be able to find the discrete logarithm of \( y \) to the base \( g \), modulo \( p \).

• The existence of one-way functions is equivalent to the conjecture \( P \neq NP \).